The CLOCK Data-Aware Eviction Approach: Towards Processing Linked Data Streams with Limited Resources

Gao, Shen ; Scharrenbach, Thomas ; Bernstein, Abraham

Abstract: Processing streams rather than static files of Linked Data has gained increasing importance in the web of data. When processing data streams system builders are faced with the conundrum of guaranteeing a constant maximum response time with limited resources and, possibly, no prior information on the data arrival frequency. One approach to address this issue is to delete data from a cache during processing – a process we call eviction. The goal of this paper is to show that data- driven eviction outperforms today's dominant data-agnostic approaches such as first-in-first-out or random deletion. Specifically, we first introduce a method called Clock that evicts data from a join cache based on the likelihood estimate of contributing to a join in the future. Second, using the well-established SR-Bench benchmark as well as a data set from the IPTV domain, we show that Clock outperforms data-agnostic approaches indicating its usefulness for resource-limited linked data stream processing.

DOI: https://doi.org/10.1007/978-3-319-07443-6_2

The CLOCK Data-Aware Eviction Approach: Towards Processing Linked Data Streams with Limited Resources*

Shen Gao, Thomas Scharrenbach, and Abraham Bernstein

University of Zurich, Department of Informatics
{shengao, scharrenbach, bernstein}@ifi.uzh.ch

Abstract. Processing streams rather than static files of Linked Data has gained increasing importance in the web of data. When processing data-streams system builders are faced with the conundrum of guaranteeing a constant maximum response time with limited resources and, possibly, no prior information on the data arrival frequency. One approach to address this issue is to delete data from a cache during processing – a process we call *eviction*. The goal of this paper is to show that data-driven eviction outperforms today's dominant data-agnostic approaches such as first-in-first-out or random deletion.
Specifically, we first introduce a method called CLOCK that evicts data from a join cache based on the likelihood estimate of contributing to a join in the future. Second, using the well-established SR-Bench benchmark as well as a data set from the IPTV domain, we show that CLOCK outperforms data-agnostic approaches indicating its usefulness for resource-limited linked data stream processing.

1 Introduction

Streams of Data have become increasingly common in the Web of Data (WoD). Constant streams of weather data, stock ticker information, tweets, bids on an auction site, and TV viewers switching channels are all examples of such streams. When processing such streams, one typically attempts to answer queries or evaluate some functions as data comes along. To that end, SPARQL-like [1] languages such as SPARQLStream [2], C-SPARQL [3], CQELS [4], TEF-SPARQL [5], and EP-SPARQL [6] were proposed to allow joining elements of the stream with each other or some rich background data set.

In contrast to static data processing systems, *stream processing systems need to be reactive*: they must process continuously arriving new data within a given set of Quality of Service (QoS) constraints. Given that latency (or the delay by which newly incoming data impacts results) is usually among these constraints, Little's law [7] 'commands' that we change from all-time semantics to one-time-semantics: data arriving after the accepted latency will not influence an answer

* The research leading to these results has received funding from the European Union Seventh Framework Program FP7/2007-2011 under grant agreement No.296126.

produced by the system. Consequently, stream processing systems have to implement measures to cope with situations where the incoming data-rate overwhelms the systems' processing capabilities – a situation we call a *stressed* system. Stress, in turn, occurs either because the constant data rate is overwhelming, hence the environment is *overloaded*, or *bursts* in the data-rate inundates the system.

Current systems typically try to avoid stress by limiting the scope of the query using a time-window – a language feature many systems support to define the *context* of a query. This solution is, however, limited to situations in which the window that is semantically relevant according to the application domain limits the arriving data to volumes that can be handled by the system. Hence, even in the light of query contexts it is easy to imagine a use case with a data rate that will overwhelm the system.

In order to deal with stress, stream processing systems can sample the incoming data, an operation called *load shedding* [8–10]. In this paper, we propose *to delete data from the caches of the operators*, as this operation can exploit the state of the operators in addition to data statistics to reduce stress. We refer to this as *eviction*, as it expels data items from the cache of operators. Both load-shedding and eviction allow maintaining the QoS constraints of a stream processing system in the light of limited resources. They do so at the cost of possibly introducing *errors*: mistakenly evicting data-items from intermediate caches that would lead to results can lower recall and even precision (when using the 'non-open world assumption' operators such as average).

This paper proposes the computationally efficient data-aware eviction strategy Clock that evicts data from a join cache based on an estimate of contributing to a join in the future. Specifically, we show that our method outperforms data-agnostic strategies such as random or First-In-First-Out (*FIFO*) using both SRBench, a standard benchmark for evaluating the performance of Linked Data stream processing systems [11], and a real-world IPTV data set. As such, the paper extends a preliminary study that showed that an omniscient eviction strategy (i.e., a strategy that could look into the future) could outperform data-agnostic scheduling strategies [12] and makes it practical due to removal of the reliance on future knowledge.

Consequently, we address the following Research Questions (RQ):

RQ 1: Real-world datasets, such as the ones in our study, can induce stress even when context limitations are present.

RQ 2: Eviction can curb memory consumption at the cost of lower recall.

RQ 3: Our Clock data-aware eviction strategy outperforms data-agnostic eviction strategies in terms of recall.

RQ 4: Clock outperforms the Least Recently Used (*LRU*) strategy, which are often used in cache management, in terms of recall.

Outline: After a conceptualization of load shedding and eviction for processing streams of data (cf. Section 2), Section 3 presents our Clock method, followed by a thorough evaluation of our research questions on two real-world data sets (cf. Section 4). After a discussion of limitations (cf. Section 5) and related work (cf. Section 6) we close with a summary of our findings (cf. Section 7).

2 System Model: A Conceptualization of Load Shedding and Eviction

A data stream processing system can be conceptualized as Processor P that continuously consumes one or more input data streams IS_i and transforms them through a series of operators O into one or more internal flows IF_j, some of which are emitted as output data streams OS_j. Hence, $P = (IS \cup OS \cup IF, O)$ can be seen as a directed graph, where the data flows along directional edges $(IS \cup OS \cup IF)$ that connect the operators $o_i \in O$, which are the nodes. All internal flows $if \in IF$ connect two operators, whilst the input streams IS_i and output streams OS_j are only connected to one operator.

In the context of the WoD the input streams IS_i typically consist of sequentially arriving data tuples of the format $< s, p, o > [t_{start}, t_{end}]$, where $< s, p, o >$ is a triple representing a fact and t_{start} / t_{end} denotes the start/end time of the triple's validity. Alternatively, when $t_{start} = t_{end}$ (i.e., the triple describes an event at time t rather than a fact) the incoming tuples can be abbreviated as $< s, p, o > t$ or, when only relative temporal order is implied by arrival time, t can be dropped. The output streams OS_j contain a continuous sequence of tuples either in the same format as the ones in the input stream or denoting bindings to a query. Note that all our considerations do not take the format of the input and output into account. Hence, our findings generalize to all stream processing systems.

System Stress This conceptualization indicates that a system can be stressed either by overwhelming the load on the operators or by inundating the bandwidth and latency constraints on the edges. This paper will focus uniquely on the former problem: It will assume that the bandwidth/latency constraints of the edges are adequate for tasks at hand. Note that operators can be overwhelemed either by time complexity (e.g., an operator that computes the factorial of large numbers) or by space complexity (e.g., a join that has to maintain a cache).

A context can curtail stress, as it allows the system ignoring nonsensical data-items and concentrating on data relevant for answering a query. A context is defined for an operator and defines which data is valid for evaluating the operator. One oftentimes used context is a time-window. Consider we want to count the audience for a certain TV channel based on a stream of events indicating which viewer switches to what TV channel. We need to know the set of data items the count is based on, i.e., the context of the operation. Prudent choices are, for example, time-based windows such as the last second (referring to the current TV ratings) or the past hour (referring to past ratings). For a detailed overview over windows and operators, we refer to [13].

Dealing with Stress We know of two approaches for dealing with stress: load shedding and eviction.

In *load shedding* the stream processing system samples the input streams and only considers part of the data. Formally, it is a sample operation $s : IS_i \mapsto \widehat{IS_i}$, where $\widehat{IS_i} \subset IS_i$. Figure 1 illustrates this for a join between stream IS_x and stream IS_y. Here stream IS_x sheds its data item x^5 at $t = 3$ by deleting it from

the considered input stream. Load shedding strategies range from deleting data at random (e.g., useful for dealing with high-frequency sensor reporting averages per time unit), via a scheduling strategy such as *FIFO*, to estimating statistics of which data to delete and which not [8–10].

In *eviction* the stream processing system removes data from the internal memory of the operators to preserve computational resources. Formally, eviction is the extension of the operators $o_i \in O$ with one or more eviction strategies $es : memory \mapsto \overline{memory}$, where $\overline{memory} \subset memory$. Figure 1 illustrates two eviction strategies: First, it 'garbage collects' items that exit the context windows win_{now} of streams IS_x and IS_y. At time $t = 2$ for both streams these are all data items, which we observed at $t = 1$, i.e., x^2 and y^2. Second, *due to the limited size of its join cache*, it decides to remove data item y^3 of stream IS_y. Note that this second strategy removes a data item which we observed at $t = 3$, i.e., a data item which would be still valid with respect to the context of stream IS_y.

This paper focuses on the impact of eviction strategies on the potential error in the resulting data. Specifically, the next section will introduce two traditional, data-agnostic evictions strategies (e.g., random eviction and *FIFO*), one based on the nature of the data (e.g., garbage collection) as well as our own Clock strategy, which relies on the likelihood of future joins.

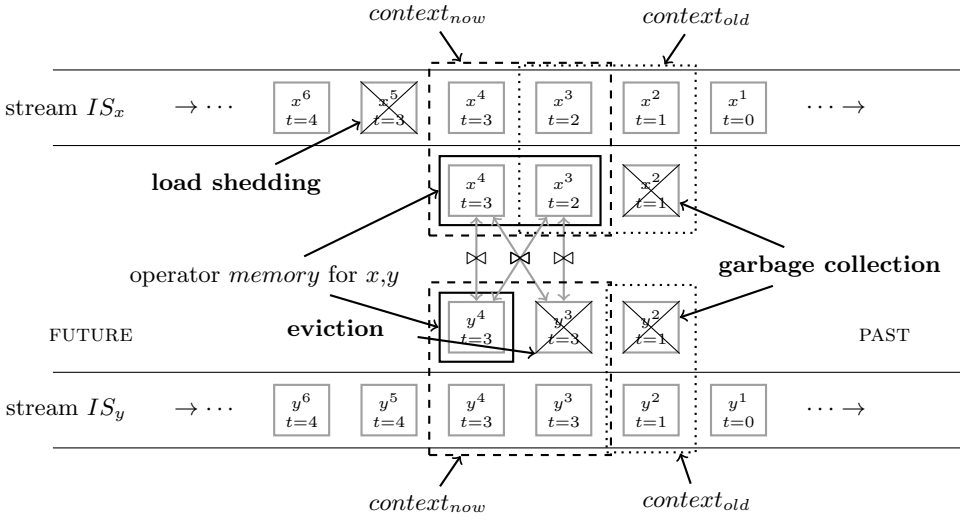


Fig. 1: Depiction of stress handling approaches in a join of two input streams. Load shedding on input stream IS_y, garbage collection on both join caches, and other (unspecified) eviction on join cache of stream IS_x. Context is shown as windows (dashed: now, dotted: past) and cache memory sizes are two items for the upper and one item for the lower stream.

3 Eviction Strategies

Eviction removes items from the internal memory (or cache) of an operator to save space. Most WoD stream processing systems extend the SPARQL algebra in order to allow evaluation of SPARQL operators on streams. As a consequence, the operators' caches typically hold candidate variable bindings. Hence, the role of the eviction strategy is to choose variable bindings to delete from the cache.

Formally, an operator's cache (or short cache) C_{op} with *limit* M and *size* N is a finite set of variable bindings $\mu_1, \ldots, \mu_N$, where $N \leq M$. We say there exists an overflow for C, if and only if $N > M$, i.e. in case the number of items in the cache exceeds the cache's limit. An eviction strategy es removes data items from a cache C such that $C' = es(C, M)$ is a cache of limit M and $|C'| \leq M$, i.e. it has no overflow. In the sequel we define different eviction strategies.

Note that in this study we consider eviction for caches of two-way-joins, i.e., joins with two join partners sharing one common join variable. We discuss possibilities for extensions to other operators in Section 5.

3.1 Baseline Eviction Strategies

In this section we succinctly introduce the four baseline or traditional eviction strategies: random, *FIFO*, *LRU*, and garbage collection.

Random eviction deletes variable bindings from a cache according to a uniform distribution $U(0, N)$ over all cache entries. To deal with cache overflow, it requires to compute $O(N - M)$ random indices to delete from the cache.

First-In-First-Out (FIFO) maintains a queue of items, where the head of the queue is deleted whenever an overflow occurs. It requires $O(N - M)$ calls to the queue. Together with random eviction *FIFO* has been adopted by today's conventional systems [10].

Least-Recently-Used (LRU), a strategy widely adopted in cache management including the SASE+ stream management system [14], extends *FIFO* by moving items to the back of the deletion queue whenever they are accessed. As with *FIFO*, handling an overflow requires $O(N - M)$ operations on the *LRU* queue.

Garbage Collection removes irrelevant data items from the operator cache. Relevancy may be determined via the context of a query. When processing TV viewership data, e.g., current viewers of a program are determined by joining the most recent program changes and user channel switches. Older channel switches by a user can, therefore, safely be garbage collected, as they are irrelevant to the query. Pure garbage collection is an incomplete eviction strategy, as it may not be able to remove enough items from the cache, when the context is not sufficiently restrictive.

Following the example of Section 2, random eviction would delete user sessions at random while *FIFO* would delete the oldest sessions – both while the session would be still valid. In a data agnostic way they 'blindly' follow their eviction strategies independent of possible future results. Garbage collection would

delete all invalid sessions. It relies on data context but ignores the performance of the item in contributing to the operation. As a metric of past performance *LRU* deletes valid sessions with no recent activity. It favors temporal recency but ignores the magnitude of a binding's past performance. In the next subsection we introduce our CLOCK approach that estimates the future likelihood of usefulness based on past performance. It extends *LRU* by considering both recency and magnitude of usefulness.

3.2 The Clock Strategy

CLOCK is a data-aware eviction strategy that considers both recency and magnitude of past usefulness of a binding to estimate the likelihood of future usefulness, which it employs as a criteria for eviction. CLOCK associates each binding with a score. Whenever an item is matched, it increases that score. When it looks for items to evict, it first depreciates the bindings' scores, and then evicts those with lower scores. Thus, the score combines a measure of recency with a measure of magnitude.

Specifically, CLOCK maintains a circular buffer cache of M slots containing the bindings μ with their associated scores w_μ and a pointer to a position p in the circle.[1] When a new data item arrives, it gets assigned an initial score $w_\mu = w^0$. If there are empty slots, it is added to one. Else the pointer depreciates the score of the item at position p using the depreciation function $dep()$. If the item's new score is lower than some threshold τ, then it gets evicted and the newly arrived binding takes its place. Otherwise, the pointer moves to the next position and repeats this procedure. Whenever a binding contributes to a join, its score gets increased by one (i.e., $w_\mu := w_\mu + 1$).

Following the example of Section 2, CLOCK increases the count whenever we observe a session activity, i.e. a user switches channels. At each point in time we decrease the count whenever we observed no activity.

Practically, we propose two different depreciation functions. The linear depreciation function $dep_{lin}(w) = w - 1$ just decreases the value of a score by one. It is associated with the threshold $\tau = 0$. Alternatively, we can depreciate exponentially with a depreciation rate ρ resulting in $dep_{exp}(w) = w * \rho$ $(0 < \rho \leq 1)$. In this case w_μ will never reach 0. Hence, we picked $\tau = 0.01$ as a threshold. We call this extended version CLOCK$_{exp}$.

In its baseline description without any extensions, CLOCK may have to circle around the cache a number of times before finding a suitable candidate for eviction. With an extension containing the currently smallest score in the cache CLOCK needs at most $O(M)$ (limit of the cache) depreciation steps to find a victim for eviction. CLOCK also requires a constant amount of additional memory (in particular M) for storing the scores w_μ of the bindings.

[1] Using a circular cache allows us to efficiently find eviction candidates by circular iterations over the buffer.

Observations: First, as mentioned, CLOCK can be seen as an extension of LRU that considers both temporal recency and past join history. The weight between these two factors can by set by adjusting ρ.

Second, the initial score w^0 reflects the degree to which we give a binding μ an initial chance to find a join partner. It should be sufficiently high, such that it has a chance to survive initially. It should be sufficiently low to ensure the timely eviction of less useful bindings. In CLOCK$_{exp}$ it determines together with ρ how dynamic the eviction strategy is.

Third, CLOCK could be easily extended to multi-way joins by using different-sized increments for partial vs. full join results.

Fourth, the CLOCK eviction strategy is founded on the following assumptions: in burst streams, eviction only takes place eventually. As a result, cache entries for which we observed no join partners could remain in the cache for a long time until eviction takes place. In an overloaded environment, there is only little chance that such items stay in the cache for long periods.

4 Evaluation

This paper argues that real-world and, hence, resource-limited WoD stream processing systems will be subject to stress even when using a use-case motivated context to limit the data that needs to be taken into consideration. To deal with stress it proposes to employ eviction – an approach that removes data from the caches of the operators of the stream processor. Specifically, it suggests to employ a data-aware eviction strategy over (more traditional) data-agnostic eviction strategies and introduces the CLOCK approach that is based on a likelihood estimate of future usefulness of an item.

To support this argumentation this section will provide empirical evidence for the research questions (RQ) we defined in Section 1:

RQ 1: Real-world datasets, such as the ones in our study, can induce stress even when context limitations are present.
RQ 2: Eviction can curb memory consumption at the cost of lower recall.
RQ 3: Our CLOCK data-aware eviction strategy outperforms data-agnostic eviction strategies in terms of recall.
RQ 4: CLOCK outperforms the Least Recently Used (LRU) strategy, which are often used in cache management, in terms of recall.

As a consequence, this section will first lay out the experimental setup (Section 4.1) and then proceed to discuss each of these research questions in turn. We first show that our data sets can be used to evaluate RQ2 and RQ3 (Section 4.2). We then evaluate these with two different experiments: first, we show the general performance of CLOCK versus other strategies (Section 4.3), then we show that we can optimize CLOCK with regards to learning its parameters (Section 4.4).

4.1 Evaluation Setup

To evaluate our research questions we built a *stream processing simulator* that allows to precisely measure, curb, and manipulate the memory consumption of the involved operators via pluggable load shedding and eviction strategies. Whilst the system does correctly identify the bindings, we call the system a simulator rather than a full-fledged stream processing systems as it was built for experimentation rather than efficient processing and lacks elements such as a query parser/optimizer.

Given our research questions, the *Key Performance Indicator (KPI) of our evaluation is recall*, which is defined as the ratio between the number of results with a given cache size to that with unlimited cache size. We disregarded the time complexity of the eviction strategy as we found that all the strategies were faster than 40 ms ($\mu = 9.45ms, var = 15.03ms$) per data item – a performance we deem sufficient for most applications.

To ensure realistic data we employ *two real-world data sets*: SRBench and ViSTA-TV. *SRBench* [11] is a well-established benchmark for assessing the semantic streaming processing engines. It comprises the LinkedSensorData, GeoNames and DBpedia.[2] Our test query focuses on the wind speed data set, because it is reported by most of the sensor stations. To simplify our experiments, we preprocessed the SRBench dataset and extracted all of the 603'642 windspeed data entries, where each triple has the format: $< sensorID, \mathsf{reports}, windSpeed > time$. Since the queries of *SRBench* were designed to benchmark the functionality of different engines, we designed a new query focused on establishing the performance of eviction strategies. The query (cf. Listing 1), defined using the TEF-SPARQL [5] semantics, aims to find sensors with similar wind speeds using a self-join on the *windSpeed* entry – an operation, where recall depends greatly on the size of join-cache employed.

```
SELECT ?sensor1, ?sensor2 FROM STREAM windSpeed
WHERE {
     ?sensor1  reports  ?windSpeed ?T1 .
     ?sensor2  reports  ?windSpeed ?T2 .
     FILTER (?sensor1 != ?sensor2) .
     FILTER(?windSpeed >= 10^^xsd:int) .
}
CONTEXT((?T1 - ?T2) <= 200^^xsd:millisecond) .
```

Listing 1: A self-join query inspired by SRBench

ViSTA-TV[3] is a FP7 financed EU project that investigates the real-time processing of TV viewership information. The data set we employed for evaluation contains anonymous IPTV viewership logs (Log) in the format $< userID, \mathsf{watches}, channelID > [t_{start_{viewer}}, t_{end_{viewer}}]$ and Electronic Program Guide (EPG)

[2] http://wiki.knoesis.org/index.php/LinkedSensorData, http://geonames.org, http://dbpedia.org

[3] http://vista-tv.eu/

data $< channelID$, plays$, programID > [t_{start_{EPG}}, t_{end_{EPG}}]$. Each data entry is annotated by a starting time stamp and an ending time stamp. A data entry is consider to be expired when the system time has passed its ending time. We used three-day's Log and EPG data, which contains 1'887'256 viewership events and 31'960 EPG entries. As defined in TEF-SPARQL [5], the query (cf. Listing 2) is a two-way join operation, which represents the use case to find all users that are currently watching a specific TV-program. To ensure that all caches were in steady state, first one third amount of data in each data set are used to 'warm up' the system and the rest are reported here.

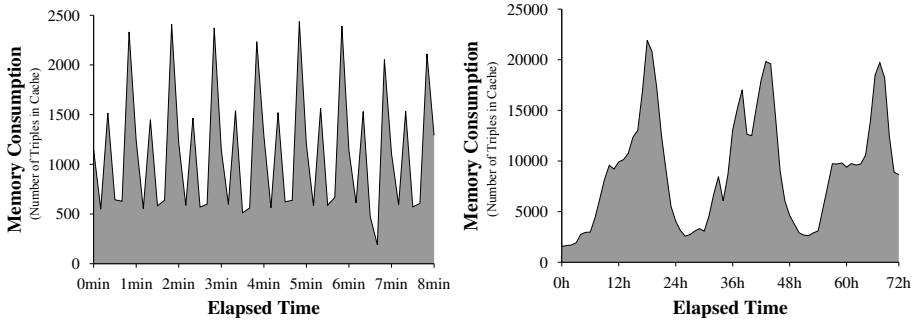
All experiments were conducted on a MacBookPro with a 2.7 GHz Intel Core i7, 16GB of RAM, and 256 GB of SSD disk space running Mac OX 10.9.1.

```
SLECT ?user, ?program FROM STREAM Log, EPG
WHERE{
     ?userID    watches ?channelID   ?T_start_viewer ?T_end_viewer .
     ?channelID plays   ?programID   ?T_start_EPG  ?T_end_EPG .
}
CONTEXT((!?T_end_viewer < ?T_start_EPG) && (!?T_end_EPG < ?T_start_viewer)).
```

Listing 2: ViSTA-TV query

4.2 RQ1: Real-world Systems are Subject to Stress

To elucidate if real-world systems are likely to be subject to stress, we graphed the cache sizes necessary to fully answer our queries for the two data sets. In other words, we assumed a system without any memory limitations and elaborated how much memory (i.e., number of triples inside cache) it needed to provide correct answers (i.e., 100% precision and recall) to our queries. Figure 2a/2b graphs 8 minutes/72 hours worth of data measured every 10 seconds/1 hour for SRBench/ViSTA-TV.



(a) SRBench (b) ViSTA-TV

Fig. 2: Fluctuations in memory consumption per time unit.

We can observe significant fluctuations in the memory size needed irrespective of the context limitations provided by the queries (e.g., the limitation on a $200ms$ window in the SRBench case). In SRBench, this is because some sensors cluster their reporting. In ViSTA-TV, the start/end times of major shows may lead to fluctuations in load.

Whilst these findings do not provide proof that systems will undergo stress conditions, they strongly indicate that real-world systems are subject to massive changes in load (hence stress). Consequently, we can argue that for any real-world system there would be a real-world data set that would overwhelm the available resources either by overloading or by burst. This, in turn, would argue for systems that are resilient against stress supporting the premise of this paper and answering *RQ1*.

4.3 RQ2-4 Eviction Results: Memory Consumption and Recall

The fact that eviction can curb memory consumption is almost self-evident. Obviously, randomly deleting data items whenever a cache-size limit is met will curb cache size. The more interesting question is what the cost of the memory limitations would be in terms of recall for a given eviction strategy.

We measured the recall gained with different cache sizes for four eviction strategies: Random, *FIFO*, *LRU*, as well as CLOCK using the linear depreciation function $dep_{lin}()$ with $\tau = 0$. Note that we did not include our prior approach [12], as it can only be used offline due to its reliance on the whole dataset; including items not yet encountered in the stream.

The results are reported in Figures 3a and 3b. All strategies were combined with garbage collection to give them the advantage of logically evicting data items that would not be used anymore.[4] We can make the following observations:

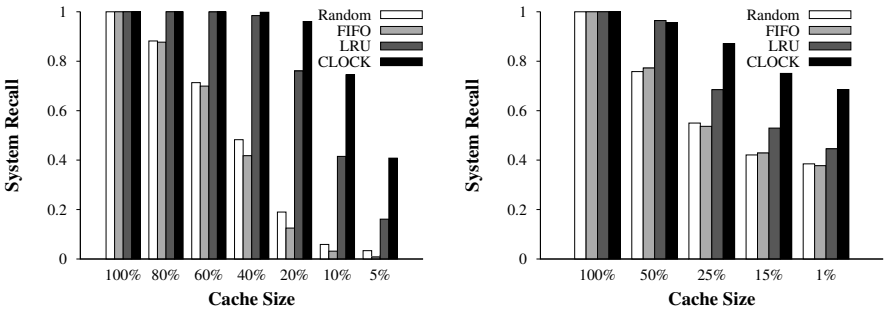
First, all strategies perform similarly with large cache sizes: systems with sufficient memory are unlikely to be stressed. Hence, eviction does not impact recall significantly.

Second, with decreasing cache size, the data-aware strategies strongly outperform Random and *FIFO* by up to 78% and 81% in ViSTA-TV and 12 and 50 times in SRBench. These results show that a *stressed* system with limited memory resources dramatically benefit from data-aware eviction strategies.

Refinement under Stress To further highlight these results, Figures 4a and 4b plot the performance results under *stressed* conditions. Hence, recalls are computed only during the number of data items per second surpassed the respective average input rate of SRBench and ViSTA-TV. The results further reinforce the above findings: CLOCK outperforms the traditionally employed *LRU* by up to 147% for SRbench and 162% for ViSTA-TV.

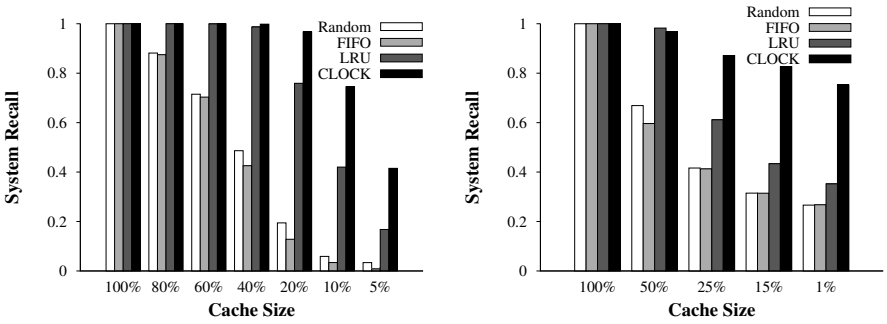
These results provide evidence to answer *RQ2*, *RQ3*, and *RQ4*. We can clearly conclude that for the given data sets data-aware methods outperform data-agnostic methods in the light of resource constraint. Further, we established

[4] Note that we cannot measure garbage collection alone, as it does not guarantee limited cache size usage.



(a) SR-Bench (b) ViSTA-TV

Fig. 3: System recall with varying cache size



(a) SR-Bench (b) ViSTA-TV

Fig. 4: Results of a *stressed* system.

that our CLOCK strategy outperforms the traditional *LRU* approach. What remains open is how robust CLOCK is towards varying depreciation functions. Specifically, how does CLOCK compare to CLOCK$_{exp}$ with different depreciation weights ρ that we discussed in Section 3.2 – a topic we will investigate in the next subsection.

4.4 Tuning Clock via Varying Depreciation Weights ρ

Different data sets may exhibit varying degrees of 'decay' in the applicability of their data items. We, hence, investigated if CLOCK could be better tuned to a data set using the depreciation functions *dep*. Specifically, we ran both CLOCK and CLOCK$_{exp}$ on our two data sets. For CLOCK$_{exp}$ we varied ρ between the following values: $\rho \in \{0.95, 0.57, 0.5.0.25\}$.

Figure 5 shows heat-maps depicting the recall for both SRBench (on the left) and ViSTA-TV (on the right). The heat-maps clearly show that the depreciation rate ρ has a profound influence in recall. For example, in SRBench, the best

	SRBench					ViSTA-TV			
	60%	*40%*	*20%*	*10%*	*5%*	*50%*	*25%*	*15%*	*1%*
LRU	0.972	0.919	0.786	0.571	0.266	0.964	0.635	0.529	0.446
CLOCK_exp 0.25	0.991	0.966	0.875	0.654	0.291	0.964	0.865	0.740	0.526
CLOCK_exp 0.5	0.998	0.989	0.936	0.770	0.413	0.932	0.818	0.762	0.683
CLOCK_exp 0.75	0.999	0.997	0.965	0.852	0.545	0.965	0.789	0.753	0.683
CLOCK_exp 0.95	0.999	0.998	0.979	0.896	0.650	0.961	0.761	0.694	0.622
CLOCK	0.997	0.992	0.967	0.875	0.567	0.956	0.799	0.751	0.685

Fig. 5: Parameter tuning for CLOCK and CLOCK_{exp} (with $\rho \in \{0.95, 0.57, 0.5.0.25\}$) on SRBench and ViSTA-TV

performance is obtained when $\rho = 0.95$. In the ViSTA-TV data set, $\rho = 0.5$ seems to provide the best performance for smaller cache sizes. Hence, in the ViSTA-TV data set it appears to better emphasize more on recent items and depreciate results faster than in SRBench. Consequently, CLOCK can be tuned according to the idiosyncrasies of a data set by choosing an appropriate depreciation rate. We hope to investigate automated tuning in the future.

5 Limitations

First, our current evaluation is limited to one operator: the join. We believe that focusing on joins for a first study made sense, as it is both the most used operator and one of the most intricate. As mentioned in Section 3, our CLOCK method could be easily extended to multi-way joins. Projections can be supported without any cache. Aggregation functions have constant memory implementations or approximations requiring investigations similar to ours. Filters are interesting, as their implementation will greatly depend on the definition of context.

Second, not neglecting the importance of throughput and latency, we deliberately focused on the very KPI that eviction will impact negatively, i.e. , recall. Other metrics will be evaluated when we implement CLOCK in real stream processing systems. Despite this limitation we believe that CLOCK's performance regarding throughput is comparable with other methods, given its low computational overhead (cf. Section 3).

A disadvantage of CLOCK is that it has to invest additional memory for storing the scores w_μ. With the same amount of memory, methods like *FIFO* and *LRU* may, hence, cache more bindings than CLOCK. However, this overhead could be minimized by implementing the score as a bitmap. Moreover, as CLOCK only needs to adjust the score for each binding, its implementation is orthogonal to other internal memory structures (e.g., a B-tree) and will not impose extra overhead on them. A next study will have to investigate the trade-off between using some memory for eviction-bookkeeping and using it only for storing bindings.

Last but not least, we will need to consider additional datasets. Whilst the two data sets considered come from two vastly different real-world applications

we believe that many more data characteristics. The compilation of more good data sets for WoD stream processing seems to be a challenge for the whole community.

6 Related Work

We discuss related work in the followings. We will first introduce different Semantic Flow Processing (SEP) systems and then discuss query processing in memory-constrained environments. Finally, we review related load shedding strategies for data stream processing.

Semantic Flow Processing Systems C-SPARQL [3] performs query matching on subsets of the information flow, which are defined by windows. The decidability of SPARQL query processing on such windows of RDF triples causes the number of variable bindings produced to be finite. However, the size of variable bindings may still become prohibitively large, e.g., when using non-shrinking semantics for aggregates [15]. For a cache of a given window size, our eviction strategies could be directly applied.

EP-SPARQL [6] and TEF-SPARQL [5] are both complex event processing systems for semantic data flows. EP-SPARQL extends the ETALIS system with a flow-ready extension of SPARQL. TEF-SPARQL distinguishes between *Events* that happen at a specific time point and *Facts* that remain valid until some events alter them. Both systems incorporate a garbage collection facility that can "prune outdated events". Since garbage collection is orthogonal to our strategies (cf Section 4.3), our findings are directly applicable to these systems.

CQELS [4] "implements the required query operators natively to avoid the overhead and limitations of closed system regimes". It optimizes the execution by dynamically re-ordering operators because "the earlier we prune the triples that will not make it to the final output, the better, since operators will then process fewer triples". This pruning does, however, not make any guarantees about the number of variable bindings created by the processors. Our methods should be directly applicable to CQELS as it provides a native implementation of the operators which contain lists of active variable bindings.

Query Processing in Memory-Constrained Environments In memory-constrained environments various techniques have been proposed to reduce the memory footprint of query planners and the number of intermediate results.

Targeting SPARQL queries Stocker et al.[16] investigated the selectivity estimates to optimize query execution. To efficiently generate alternative query plans, [17] proposed a branch-and-bound to enumerate join plans for left-deep processing trees. This method requires less memory as it prunes the search space during enumeration. Our eviction strategies are designed for caches and assume a given query execution plan.

Regarding multiple aggregate queries over stream data Naindu et al. [18] proposed a new hash model for estimating the cost for intermediate aggregates. This method groups common attributes of related queries and reduces overall

memory usage. Based on this new model, they also proposed a greedy heuristic to generate the execution plan. Our eviction strategies are designed for general semantic streaming systems that perform not only aggregate query, but also other kinds of queries.

In a XML processing system the memory consumption for XML processing can greatly exceed the actual file size. Therefore, an entire XML document may not fit into main memory. In [19] the authors proposed a method that analyses XQuery to identify and extract only useful attributes form XML documents during compilation to reduce the file size. Our eviction strategies deal with semantic data, where it is straightforward to identify useful attributes from input stream. Meanwhile our strategies are also applicable to projected variable bindings.

Load Shedding Load shedding has been applied to information flow processing. Approaches like [8–10] perform load shedding by dropping tuples from the stream, i.e., dropping data instead of variable bindings.

In [10] the authors proposed to insert a "drop operator" into the query execution plan, which automatically decides where, when and how to perform load shedding. Regarding how to perform load shedding they proposed a random method as a baseline and a "semantic method" which decides whether to retain a data entry based on estimating its impact on QoS. In addition to their approach, our strategies also take into account the time a data entry has resided in memory. Similar to [10], [9] also proposed a special operator that decides where and when to drop unprocessed data by using statistical methods. However, [9] only focuses on aggregate queries.

SASE+ [14] employs an automata-based matching approach. Similar to our case of caching variable bindings, SASE+ stores automata states. The authors do apply some eviction strategy. However, their strategy is based on a deterministic approach that is similar to *FIFO* and *LRU* in our baseline approaches.

Finally, Das et al. [8] propose a simple equi-join on two incoming streams and to evict tuples that are unlikely to find a join partner. However, this method works only with a sliding window and with a single equi-join of two streams. Our approaches could be applied on caches for any kind of join.

7 Conclusion and Outlook

In this paper, we presented our data-aware eviction strategy Clock, which addresses stress in WoD stream processing systems. We found that stress in terms of overloading and bursts occurred in our two real-world datasets. In addition, Clock and its variant Clock$_{exp}$ outperform the often-used *LRU* strategy by factors between 1.5 and almost 3 and *FIFO* strategy by even higher factors.

The next step in our investigation will be to implement these strategies in a real stream processing system to study the trade-off between recall and other KPIs such as latency and throughput with different data sets. Whilst our work is only a first step in investigating resource-limited stream processing, we believe it pursues an important direction that sets the expectation for the real-world usage of such systems.

Acknowledgments We would like to thank Khoa Nguyen for all his earlier work on this topic and Daniel Spicar for his constructive advice.

References

1. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. Technical report, The World Wide Web Consortium (W3C) (2011)
2. Calbimonte, J.P., Corcho, O., Gray, A.: Enabling ontology-based access to streaming data sources. In: Proc. ISWC. (2010) 96–111
3. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: C-SPARQL: A Continuous Query Language for RDF Data Streams. International Journal of Semantic Computing (1) (2010) 3–25
4. Le-Phuoc, D., Dao-tran, M., Parreira, J.X., Hauswirth, M.: A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. In: Proc. ISWC. (2011) 370–388
5. Kietz, J.U., Scharrenbach, T., Fischer, L., Bernstein, A., Nguyen, K.: TEF-SPARQL: The DDIS query-language for time annotated event and fact Triple-Streams. Technical report, University of Zurich, Department of Informatics (2013)
6. Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: EP-SPARQL: a unified language for event processing and stream reasoning. In: Proc. WWW. (2011) 635–644
7. Little, J.D.C.: A proof for the queuing formula: L= λ w. Operations Research **9**(3) (1961) pp. 383–387
8. Das, A., Gehrke, J., Riedewald, M.: Approximate join processing over data streams. In: Proc. SIGMOD, New York, New York, USA (2003) 40–51
9. Babcock, B., Datar, M., Motwani, R.: Load shedding for aggregation queries over data streams. In: Proc. ICDE. (2004) 350–361
10. Tatbul, N., Çetintemel, U., Zdonik, S., Cherniack, M., Stonebraker, M.: Load Shedding in a Data Stream Manager. In: 29th International Conference VLDB. (2003) 309–320
11. Zhang, Y., Duc, P.M., Corcho, O., Calbimonte, J.p.: SRBench : A Streaming RDF / SPARQL Benchmark. In: Proc. ISWC. LNCS (2012) 641–657
12. Nguyen, K., Scharrenbach, T., Bernstein, A.: Eviction Strategies for Semantic Flow Processing Systems. In: Proc. SSWS. (2013)
13. Cugola, G., Margara, A.: Processing flows of information. ACM Computing Surveys **44**(3) (June 2012) 1–62
14. Diao, Y., Immerman, N., Gyllstrom, D.: Sase+: An agile language for kleene closure over event streams. Technical report, University of Massachusetts Amherst, Department of Computer Science (2008)
15. Barbieri, D., Braga, D., Ceri, S.: Incremental reasoning on streams and rich background knowledge. In: Proc. ESWC. (2010) 1–15
16. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: Sparql basic graph pattern optimization using selectivity estimation. In: Proc. WWW. (2008) 595–604
17. Bowman, I., Paulley, G.: Join enumeration in a memory-constrained environment. In: Proc. ICDE. (2000) 645–654
18. Naidu, K., Rastogi, R., Satkin, S., Srinivasan, A.: Memory-constrained aggregate computation over data streams. In: Proc. ICDE. (2011) 852–863
19. Marian, A., Siméon, J.: Projecting xml documents. In: Proc. VLDB. (2003) 213–224